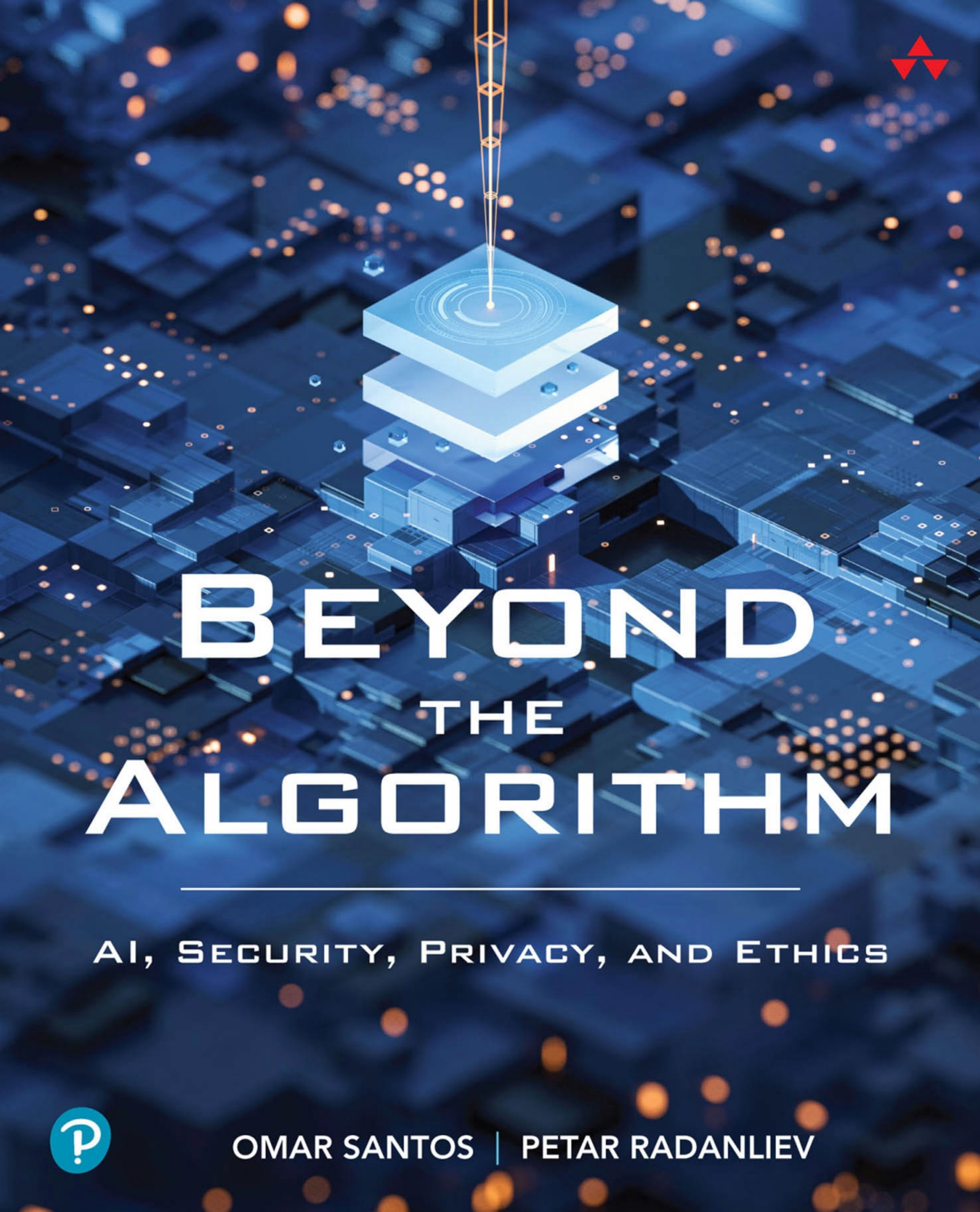

Beyond the Algorithm

AI, Security, Privacy, and Ethics

OMAR SANTOS | PETAR RADANLIEV

BEYOND THE ALGORITHM

```
z_mean = Dense(latent_dim)(hidden_encoder)
z_log_var = Dense(latent_dim)(hidden_encoder)

# Reparameterization trick
def sampling(args):
    z_mean, z_log_var = args
    epsilon = tf.random.normal(shape=(tf.shape(z_mean)[0], latent_dim))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon

z = Lambda(sampling)([z_mean, z_log_var])

# Decoder
hidden_decoder = Dense(intermediate_dim, activation='relu')
output_decoder = Dense(input_dim, activation='sigmoid')

z_decoded = hidden_decoder(z)
outputs = output_decoder(z_decoded)

# VAE model
vae = Model(inputs, outputs)

# Loss function
reconstruction_loss = MeanSquaredError()(inputs, outputs)
kl_loss = -0.5 * tf.reduce_sum(1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_
var), axis=-1)
vae_loss = tf.reduce_mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)

# Compile and train the VAE
vae.compile(optimizer='adam')
vae.fit(x_train, x_train, epochs=50, batch_size=128, validation_data=(x_test, x_
test))

# Generate new samples from the latent space
n = 15
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

grid_x = np.linspace(-4, 4, n)
grid_y = np.linspace(-4, 4, n)[::-1]

for i, yi in enumerate(grid_y):
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi]])
        x_decoded = output_decoder(hidden_decoder(z_sample))
        digit = x_decoded[0].numpy().reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.axis('off')
plt.show()
```

The program in Example 3-2 creates a simple VAE with a 2D latent space, trains it on the MNIST dataset, and generates new handwritten digits by sampling from the latent space. You can modify the latent_dim variable to change the dimensionality of the latent space or adjust the intermediate_dim variable to change the size of the hidden layers in the encoder and decoder networks.

After training the VAE, the program generates a grid of new samples from the latent space by sampling points in a two-dimensional grid. It decodes these points using the trained decoder network and visualizes the resulting handwritten digit images in a grid format using Matplotlib. The generated images should resemble handwritten digits, demonstrating the capabilities of VAEs in learning the underlying data distribution and generating new samples that are like the training data.

Note

You can experiment with different VAE architectures or training parameters to see how they affect the quality of the generated samples. Additionally, you can try applying VAEs to other datasets or problems, such as image generation for other domains, text generation, or even learning meaningful latent representations for various tasks like clustering or classification.

Autoregressive Models

Autoregressive AI models are a class of generative AI models that predict future values in a sequence based on their past values. They are designed to capture the dependencies between data points in a time series or sequential data. Autoregressive models have also been widely used in various fields, including time series forecasting, natural language processing, and image synthesis.

The primary idea behind autoregressive models is to express the value of a data point at a particular time step as a linear combination of its past values. This is done along with a noise term. The number of past values (otherwise known as *lag*) used in the model determines the order of the autoregressive model.

In other words, an autoregressive AI model of order *p (AR(p))* uses the *p* previous values to predict the current value.

There are several types of autoregressive models, each with its own strengths and weaknesses as outlined in Table 3-4.

Table 3-4 The Pros and Cons of Different Types of Autoregressive Models

Model Type	Pros	Cons
Autoregressive (AR) Models	Simple, easy to understand and implement; captures linear relationships between past and present values.	Assumes a linear relationship; may not capture complex patterns or seasonality.
Moving Average (MA) Models	Captures linear relationships between past errors and present values; smooths out noise in the data.	Assumes a linear relationship; may not capture complex patterns or seasonality.
Autoregressive Integrated Moving Average (ARIMA) Models	Combines AR and MA models; handles nonstationary data through differencing; captures both past values and error terms.	Assumes a linear relationship; may require significant tuning of parameters; may not capture complex patterns or seasonality.
Seasonal Decomposition of Time Series (STL) Models	Decomposes data into components; handles seasonality; captures both linear and nonlinear relationships in individual components.	Requires multiple models for each component; may be computationally expensive.
Neural Autoregressive Models	Leverages deep learning techniques; captures complex patterns and nonlinear relationships; can handle large amounts of data and high dimensionality.	Requires large amounts of data for training; may be computationally expensive; may require significant tuning of model parameters.

The training process for autoregressive models typically involves optimizing the model's parameters to minimize the prediction error. For linear autoregressive models, the parameters can be estimated using methods such as least squares or maximum likelihood estimation.

For neural autoregressive models, gradient-based optimization techniques such as stochastic gradient descent or adaptive learning rate methods are commonly used.

Several advantages of autoregressive AI for modeling sequential data exist, including the following:

- **Interpretability**: Generally, autoregressive AI models are more interpretable than other generative models because their parameters directly capture the relationships between data points.
- **Flexibility**: Autoregressive AI modeling can be extended to handle various types of data, including nonstationary and seasonal time series.
- **Scalability**: Autoregressive AI modeling can be scaled to large datasets using efficient optimization algorithms and parallel computing techniques.

Autoregressive models have been successfully applied to a wide range of real-world problems, including the following:

- **Time Series Forecasting**: Predicting future values of stock prices, energy consumption, or weather variables.
- **Natural Language Processing**: Modeling the probability distribution of words or characters in a text, enabling tasks such as text generation and machine translation.
- **Image Synthesis**: Generating realistic images by modeling the dependencies between pixels in an image.

Restricted Boltzmann Machines (RBMs)

Restricted Boltzmann machines are a class of energy-based generative models originally developed for unsupervised learning tasks. An RBM consists of two layers of nodes: a visible layer and a hidden layer.

Note

Energy-based means that the model associates an "energy value" with each possible configuration of visible and hidden nodes. The goal of the model is to learn a probability distribution over the input data that minimizes this energy value. RBMs played a significant role in the development of deep learning techniques in the late 2000s. Energy-based generative models are a class of generative models that associate an energy value with each possible configuration of the model's variables. The primary goal of these models is to learn a probability distribution over the input data that assigns low energy values to plausible data samples and higher energy values to less likely or implausible samples. By doing so, energy-based models capture the underlying structure and dependencies present in the data.

Energy-based generative models consist of two main components:

- **Energy Function**: This function assigns an energy value to each configuration of the model's variables, which can include observed data and latent variables. The energy function is typically parameterized by a set of learnable parameters, which are adjusted during the training process.
- **Partition Function**: The partition function is a normalization term used to convert the energy values into a valid probability distribution. It is calculated as the sum (or integral, in the case of continuous variables) of the exponentiated negative energy values over all possible configurations of the model's variables.

Training an energy-based generative model involves adjusting the model's parameters to minimize the energy values for the observed data samples while ensuring that the partition function remains tractable. This can be achieved by optimizing an objective function, such as the log-likelihood of the data or a variational lower bound on the log-likelihood.

Energy-based generative models have been used for various tasks, including unsupervised learning, density estimation, and generative modeling, across various domains such as computer vision, natural language processing, and reinforcement learning.

The following papers provide a comprehensive understanding of energy-based generative models. Although some of these papers date back several years, they remain valuable resources for reference and understanding the historical development of energy-based generative models in the field of machine learning and artificial intelligence:

- Y. LeCun and F. J. Huang, "Loss Functions for Discriminative Training of Energy-Based Models," *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics* 3 (2005): 206–13, https://proceedings.mlr.press/r5/lecun05a/lecun05a.pdf.
- G. E. Hinton, "Training Products of Experts by Minimizing Contrastive Divergence," *Neural Computation* 14, no. 8 (2002): 1771–1800, https://www.mitpressjournals.org/doi/abs/10.1162/089976602760128018.
- G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, 313, no. 5786 (2006): 504–507, https://www.science.org/doi/10.1126/science.1127647.
- P. Smolensky, "Information Processing in Dynamical Systems: Foundations of Harmony Theory," in D. E. Rumelhart and J. L. McClelland, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1 (MIT Press, 1986): 194–281, https://web.stanford.edu/group/pdplab/pdphandbook/handbookch6.html.
- R. Salakhutdinov and G. E. Hinton, "Deep Boltzmann Machines," *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics* 5 (2009): 448–55, http://proceedings.mlr.press/v5/salakhutdinov09a/salakhutdinov09a.pdf.

RBMs have been widely used for feature learning, dimensionality reduction, and pretraining deep learning models, such as deep belief networks and deep autoencoders.

Tip

Feature learning and dimensionality reduction are two closely related unsupervised learning techniques aimed at extracting meaningful representations from raw data. Feature learning, also known as *representation learning*, involves discovering high-level, abstract features in the data that can be used for various tasks, such as classification or clustering. Dimensionality reduction, on the other hand, focuses on reducing the number of variables in the data while preserving its underlying structure or relationships between data points. Both techniques help in transforming complex data into more manageable and interpretable forms, ultimately improving the performance of machine learning models and enhancing data visualization.

The visible layer corresponds to the input data, while the hidden layer captures the underlying features or representations of the data. Each node in the visible layer is connected to every node in the hidden layer, but nodes within the same layer are not connected to each other, hence the term *restricted*. This is illustrated in Figure 3-5.

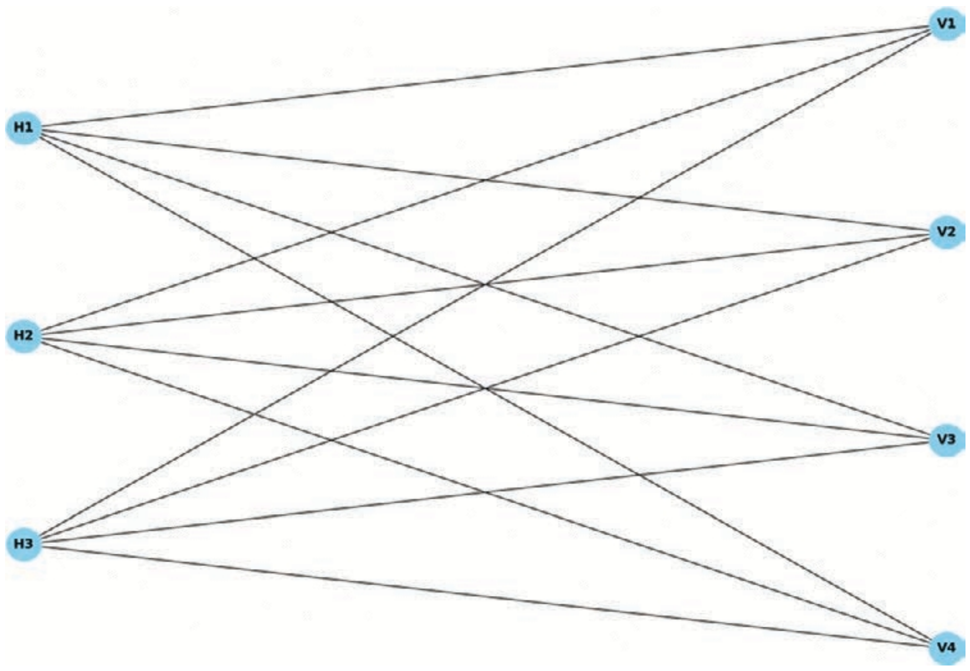

Figure 3-5
Visualization of RBMs

The code to make the graphic in Figure 3-5 is available in my GitHub repository at https://github.com/santosomar/responsible_ai/blob/main/chapter_3/RMB_visualization.py.

The training process of RBMs involves adjusting the weights between the visible and hidden nodes to minimize the energy of the system. This is done by maximizing the likelihood of the input data, which corresponds to minimizing the free energy of the model. The learning algorithm commonly used to train RBMs is called contrastive divergence (CD).

Contrastive divergence involves two main steps: a positive phase and a negative phase. During the positive phase, the input data is used to update the hidden node activations. In the negative phase, the model generates new data samples by performing Gibbs sampling, a Markov Chain Monte Carlo technique.

Note

Gibbs sampling is a Markov Chain Monte Carlo (MCMC) technique used for sampling from a multivariate probability distribution when direct sampling is difficult. It is particularly useful when the joint distribution is complex, but the conditional distributions are relatively simple and easy to sample from. Gibbs sampling is widely used in various applications, including Bayesian inference, topic modeling (e.g., Latent Dirichlet Allocation), and training certain generative models, such as RBMs and deep Boltzmann machines (DBMs).